METAHEURISTICS FOR THE WEIGHTED SQUARED TARDINESS PERMUTATION FLOWSHOP SCHEDULING PROBLEM

by

Andreia Fernandes da Silva

Dissertation for Master in Modeling, Data Analysis and Decision Support System in Optimization

Supervised by

Jorge Valente

Jeffrey Schaller

2016

Biographical introduction of the candidate

My name is Andreia Silva and I was born in1992. My home town is Porto and I lived all my life in the city where I was born.

When I was a kid I wanted to be an architect. I liked to draw and make plants of my dream house but then I met Math's and I fell in love.

Since then, my dream was to make money. Who doesn't have the same dream?

I wanted to deal with money and come up with strategies that could save companies' lives.

I finished high school and I tried out for Sport University, I wasn't always so sure of what I wanted. I passed the try-outs and I signed up to College in management course. That doesn't make sense, I know, but then something happened. I fell in love again. That was definitely what I wanted to do.

I finished my course knowing that I was in the right place.

I decided then to continue my studies and reach a new accomplishment so I entered in MADSAD course to become more aware of what I could do to save companies' lives.

Biografia introdutória do candidato

O meu nome é Andreia Silva e nasci em 1992. Nasci no Porto, em Massarelos e sempre vivi, toda a minha vida no Porto, na Maia.

Quando era mais nova, sonhava em ser arquitecta. Chegava mesmo a desenhar e a fazer plantas da minha casa de sonho mas depois conheci a Matemática e apaixonei-me.

A partir desse momento, o meu sonho foi ganhar dinheiro. Quem não tem o mesmo sonho?

Eu queria lidar com dinheiro e arranjar estratégias ou alternativas que pudessem salvar a vida das empresas.

Quando terminei o secundário, fiz os pré-requisitos na Faculdade de Desporto da Universidade do Porto, nem sempre tive a certeza do que realmente queria seguir. Passei nos pré-requisitos mas quando chegou a altura de me candidatar à faculdade, candidatei-me no curso de Gestão. Não faz qualquer sentido, mas algo aconteceu. Apaixonei-me novamente. Era sem dúvida alguma ali que eu queria estar.

Acabei o meu curso sabendo que estava no lugar certo.

Decidi depois continuar os meus estudos e entrei num mestrado, no curso MADSAD (Modelação Análise de Dados e Sistemas de Apoio à Decisão) na Faculdade de Economia da Universidade do Porto para conseguir ter outra sensibilidade do que poderia fazer para salvar algumas empresas.

Acknowledgements

Well, I am so grateful for having so many people to thank for. Obviously not everyone here mentioned helped me directly in this process but they do part of my life and, in one way or another, they encouraged me one this academic phase.

First of all, I would like to thank my adviser, Professor Jorge Valente for the support, stimulus and for the sense of humor. It would all be much difficult without his help and guidance. I thank also to Jeffey Schaller who was involved in all process.

I would like to thank next to my course director, Professor João Gama for the incentive to keep working.

To finish my academic acknowledgments, I would like to thank Professor Hugo Alonso who inspired me as a student. Always watchful, perfectionist and with whom I had the pleasure to work and learn. Professor Alonso, a big thank to you.

Then, to my family, my support and where I refuged every time I needed and who never let me down.

To my bests friends for the companionship, for the moments of fun, for the hardest moments and for letting me believe that friendships can last a life time. To Morais, Néné, Paulinha, Tono, Luís, Pombo and Top.

I thank too Pipa for the chats, for the support and for the sincere friendship.

To my paternal family for the craziness. To my grandfather for the infinitive intelligence, for showed me what real strength was and whose I am tremendously proud. To my grandmother for the happiness. To my uncles and cousin, Tété, Tone and Didi for the sympathy. To Nuno for the example and inspiration. To my Godfather for the genuine craziness and good mood. To Jú and Bi for the concern and constant presence.

To my maternal family for the love, union e for helped me growing. To my grandmother, my queen not only for the generosity as for the philosophy of life that characterizes her. To my great-aunt for her young spirit. To my uncle Paulinho for the jokes and for my aunt Rosa for the humility. To my little cousin Mariana for the innocence and genuineness. To my Godmother for everything, for being the best and for

always being there. To my uncle Jaime for the music, for the sense of humor e for our chats. To my little cousin Pedrinho, for the jokes, hugs and intelligence. To my cousin Dani who was always like a sister to me, for the chats and concern. To my aunt Elsa who always believed in me and made me persuade my way, for the companionship and for being amazing. To my uncle Paulo, for the help, advices and guidance. To my cousin Sara for the best sense of humor ever, for the innocence and cuteness. To my goddaughter and cousin Elsinha for being the cute person in the world and for the affectionate hugs.

To my best friend, confident and boyfriend, Pedro Nuno. To you, I thank everything, you were always there. You lift me up every time and advised when necessary. I admire you e thank you for never doubt of me. To his parents for the house and for the affection.

Finally to my parents. To them, I dedicate everything. People with amazing values. Better parents is not possible. I own them everything. To my mom Anabela, for being the best person in the world. For giving the best hugs and for believing be. For making me the person I am today. To my father José, for the stubbornness, affection and for the constructive flaks that made me try to be better every day. For the support and infinitive knowledge. To you both, the biggest thank ever!

Agradecimentos

Bem, estou grata por ter tanta gente a quem agradecer. Obviamente que nem todos os que aqui são mencionados me ajudaram directamente na concretização desta etapa mas fazem parte da minha vida e, de uma maneira ou de outra, incentivaram-me nesta fase do meu percurso académico.

Em primeiro lugar gostaria de agradecer ao meu orientador, Professor Jorge Valente pelo apoio, pelo estímulo e pelo sentido de humor. Tudo teria sido bem mais complicado sem a sua ajuda e orientação. Agradeço também ao Jeffrey Schaller que esteve também envolvido em todo o processo.

De seguida, gostaria de agradecer ao meu director de curso, Professor João Gama pelo incentivo constante em continuar a trabalhar.

Para terminar os agradecimentos académicos, gostaria de agradecer ao Professor Hugo Alonso que me marcou muito como aluna. Professor sempre atento, perfeccionista e com quem tive o prazer de aprender e trabalhar. Professor Alonso, um grande obrigada.

Depois, à minha família, o meu suporte e onde sempre que necessário me refugiei e que nunca me falhou com nada.

Aos meus melhores amigos pelo companheirismo, pelos momentos de diversão, pelo apoio nos momentos mais complicados e por me fazerem acreditar que há amizades que podem durar uma vida inteira. Ao Nuno, ao Néné, à Paulinha, ao Tono, ao Luís, ao Pombo e ao Top.

Agradeço também à Pipa pelas conversas, pelo apoio, pelos desabafos e pela amizade sincera.

Depois, à minha família, o meu suporte e onde sempre que necessário me refugiei e que nunca me falhou com nada.

À minha família paterna pela maluqueira. Ao meu avô pela inteligência infinita, por me ter mostrado o que realmente é ter força e de quem estou tremendamente orgulhosa. À minha avó pela alegria. Aos meus tios e primo, Tété, Toni e Didi pela simpatia. Ao

Nuno pelo exemplo e inspiração. Ao meu Padrinho pela genuína maluqueira e boa disposição. Aos meus tios Jú e Bi pela preocupação e presença constante.

À minha família materna pelo amor, pela união e por me ter ajudado a crescer. À minha avó que é a minha rainha não só pela generosidade como pela filosofia de vida que tanto a caracteriza. À minha tia-avó Rosete pelo seu espírito jovem. Ao meu tio Paulinho pelas brincadeiras e à minha tia Rosa pela humildade. À minha priminha Mariana pela inocência e genuinidade que tanto me cativam. À minha Madrinha por tudo, por ser a melhor do mundo e por ter sempre estado presente. Ao meu tio Jaime pela música, pelo sentido de humor e pelas nossas conversas. Ao meu priminho do coração Pedrinho, pelas brincadeiras, abraços e inteligência. À minha prima Dani que sempre foi como uma irmã para mim, pelas conversas e pela preocupação. À minha tia Elsa que sempre acreditou em mim e me fez seguir um caminho que acreditava ser o melhor, pelo companheirismo e por aquela maneira de ser fantástica. Ao meu tio Paulo, pelas explicações, pelos conselhos e orientações. À minha prima Sara pelo sentido de humor mais apurado que conheci, pela inocência e pela fofura. À minha afilhada e prima Elsinha por ser a pessoa mais querida do mundo e pelos abraços mais carinhosos de sempre.

Ao meu melhor amigo, confidente e namorado Pedro Nuno. A ti agradeço-te por tudo, sempre estiveste lá e nunca me falhaste com nada. Levantaste-me do chão quando foi preciso mas também me soubeste chamar à atenção nas alturas certas. Admiro a excelente pessoa que és e obrigada por nunca teres duvidado de mim e por me teres feito seguir em frente, obrigada. Aos pais dele por me terem dado uma segunda casa e pelo carinho.

E finalmente aos meus pais. A eles dedico tudo. Pessoas de valores extraordinários e que admiro acima de tudo. Posso dizer com toda a certeza que me saiu a sorte grande e que melhores pais seria impossível. Devo-lhes tudo. À minha mãe Anabela por ser a melhor pessoa à face da terra. Por dar os melhores abraços do mundo. Por acreditar em mim. Por fazer de mim a pessoa que sou. Por me dar a oportunidade de ter um modelo a seguir. Ao meu pai José, por ser como é. Pela teimosia. Pelo carinho. Pelas críticas construtivas que me fizeram tentar ser melhor todos os dias. Pelo apoio e pelo conhecimento infinito. A vocês os dois, o maior obrigada de sempre!

Abstract

The main issue for a company to endure is the profit. However, the processes to achieve profit are quite complex. The essential is to leave all the clients satisfied and see them as a weapon to the company's success. Making the clients satisfied isn't only to provide them a service or a product with the highest quality, but also to provide that same service or product as fast as it is possible.

This work deals with the cost associated with time. Metaheuristics were used to generate a schedule for a set of jobs, considering a certain number of machines, in order to minimize the costs associated with delayed deliveries.

After analyzing the computational results, we concluded that the two metaheuristic proposed achieved solutions with high quality in feasible computational run time. Small and medium sized problems were considered, and for these the metaheuristics used proved to be very efficient.

Key-words: Permutation flowshop, weighted squared tardiness, metaheuristics, iterated local search, steady-state genetic algorithm.

Resumo

O princípio básico para que uma empresa perdure é que tenha lucro. Todo o processo até chegarmos ao lucro, é já mais complexo. O essencial será sempre deixar os clientes satisfeitos e vê-los como arma de sucesso da empresa. Deixar os clientes satisfeitos não tem apenas a ver com a qualidade do serviço prestado ou do produto vendido mas também com a rapidez de resposta a um pedido/encomenda.

Este trabalho tem por base o custo associado ao tempo. Foram usadas meta heurísticas para sequenciar um conjunto de trabalhos tendo em consideração um determinado número de máquinas para que desta forma a empresa tenha o menor custo possível devido a atrasos na entrega.

Depois de analisar os resultados computacionais, podemos concluir que as duas meta heurísticas aqui propostas, alcançaram soluções com boa qualidade e num tempo computacional razoável. As instâncias consideradas tinham uma dimensão pequena a média, e neste contexto as meta heurísticas provaram ser eficientes.

Palavras-chave:

Permutação em *flowshop*, atraso com peso ao quadrado, meta heurísticas, *iterated local search*, *steady-state genetic algorithm*.

Table of contents

List of tables

Chapter I- Introduction

1.1- Motivation

Ever since we start to know what money is and became aware of the notion of time, we hear the expression “time is money”.

The problem is that companies seem to forget how precious time is and how a simple thing as time can save the whole company.

With this work, but mostly with this theme, it is easy to understand the importance of having orders ready in time, but that doesn’t necessarily means that the orders have to be ready when the client wants them. Everything that has to do with business implies money, costs and earnings.

This is like a game to play. Is important to know when we have to rush and when we have to slow things down to combine all the orders that we have.

With this work it is possible to optimize the time we have available and to minimize the tardiness we will have.

Optimizing the time available and minimizing the tardiness in responding to a client’s order, increases the possibility of fewer costs and lower losses. Thus, optimizing the use of time it is possible to make more money.

For this work we have chosen a scheduling problem with permutation flowshop because we considered a factory environment. This way of production is very common, not only in a factory environment, but also in other settings.

On the other hand, permutation flowshop is mandatory in some lines of production but even when it is not compulsory, is often used because it greatly simplifies the assembly line. The machines are never turned off and it is not necessary to change one job from a machine to another that is not the next machine where the job is being done.

1.2- Proposition

This work, as it was mentioned before, contemplates a permutation flowshop scheduling problem with weighted squared tardiness costs.

This specific problem can be defined as a set of jobs that has to be processed on a set of machines. This scheduling has a restriction. All jobs have to pass through all the machines, and in the same order. The name permutation flowshop comes from this latter restriction. From the moment one job starts processing in one machine, it cannot be interrupted, so preemptions are not allowed. All machines are always available.

Formally in this problem there is a set N= {1, 2,…, n} of n independent jobs that need to be processed in a set M={1, 2,…, m} of m machines. Job j, j∈ N involves a processing time p_{ij} on machine i, i ∈ M and also has a weight w_j and a due date d_j.

The processing time measures the time that one job needs until it's ready on a certain machine, the weight measures the importance of a certain job and at last, the due date is the date on which the job supposed to be delivered. It is known in this problem that all machines are available at time zero.

C_{ij} is the completion time of job j, j∈ N on machine i, i ∈ M. Also, let [j] denote the job that is scheduled in position j. Thus, $C_{1\,[0]} = 0$ since all machines are available at time zero.

Consequently, $C_{1\,[j]} = C_{1[j-1]} + p_{1\,[j]}$ and $C_{k\,[j]} = \max\{ C_{k-1[j]}, C_{k\,[j-1]}\} + p_{k\,[j]}$, for k = {2,3,…,m}. Finally, the completion time at which the job j finishes processing on the last machine is also denoted by C_j, with $C_j = C_{mj}$.

For a given schedule, the tardiness of job j is defined as $T_j = \max\{C_j - d_j ; 0\}$. The main goal of this work is to find a schedule that minimizes the sum of the weighted squared tardiness values.

The sum of the weighted squared tardiness values is given by $\sum_{j=1}^{n} W_j\, T_j^2$. This expression combines the tardiness of one job and the weight of that job.

Both tardiness and earliness in a company represents costs for that company. In this work we will only consider tardiness as objective of study. The reason why we will only

work with tardiness costs has to do, on the one hand ,with the philosophy of some administrators of the companies. Several believe tardiness costs have more importance than earliness costs so delivering an order late is worse than delivering it early. On the other hand, sometimes earliness costs are negligible.

The squared tardiness is used since the customer's discontent tends to increase quadratically with the tardiness. On the other hand, squared tardiness avoids situations in which only one or a few jobs contribute the majority of the cost.

We choose to use square tardiness as the objective function and not linear tardiness or maximum tardiness alternatives. None of these three measures is essentially better. Indeed, depending on the problem to be studied and the goals to achieve, each of these criteria can be appropriate.

A maximum tardiness criterion, as the name regards, is used when the main goal is to avoid a quite large delay. As explained in (Sun, Noble and Klein 1993), maximum tardiness concentrates on the job with the largest delay and ignores the tardiness in the remaining jobs if exists. As the main goal in this work is to minimize the sum of the squared tardiness of all jobs resulting in discontent of customers, it is preferable to use squared tardiness over the maximum tardiness criterion. That does not mean that square tardiness is always better than maximum tardiness. It just means that for this type of problem, squared tardiness is more suitable than maximum tardiness regarding the main goal.

On the other hand, in linear tardiness the distribution of the tardiness is immaterial. As it was mentioned before, in this work, we want that a job or a small set of jobs don't contribute the majority of the cost. As mentioned in (Hoitomt, Luth and Pattipati 1990) and (Thomalla 2001), if the tardiness increases, the consequences won't increase accordingly. That doesn't happen in squared tardiness. Here, the consequences of a job do increase with tardiness. And as it was mentioned earlier, to justify the use of squared tardiness, the discontent of a customer increases quadratically with tardiness (as it was defended in (Taguchi 1986), in contrast of what happens with linear tardiness.

In this work, the order of the jobs is the same in all machines not just because it is easier to compute, but also because in several real settings it isn't feasible to change the order

of jobs on the machines. Since we want this to be as applicable as possible, we adjust the procedure to real life.

We will be present and analyze efficient metaheuristics. The metaheuristics that will be used are Iterated Local Search (ILS) and Steady-State Genetic Algorithm (SSGA).

1.3- Contents

As it was said before, this work deals with a permutation flowshop problem with weighted squared tardiness. The main goal of this work it to understand the behavior of metaheuristics when applied to this kind of problems in order to minimize the tardiness of client's orders and consequently minimize the costs for the company. During the different chapters we will present the development of this investigation and further the results that we achieved and the final conclusions. What which chapter presents is explained below.

In Chapter II we introduce some of the articles that we reviewed before writing this work. As it will be mentioned next, there aren't many authors that studied this type of problem with the same restrictions, but there are similar themes in articles that are very helpful to conclude and understand the behavior of the different metaheuristics. There is one particular work (Costa 2015) that will be used in a comparison with our results, since it studies the same problem using different methods.

Next, in Chapter III we describe the metaheuristics used, and the multiple versions that were tested for each one. All the pseudo-codes are also presented and described in this chapter.

Chapter IV includes the presentation of the computational results achieved during this investigation. First we introduce the instances and the programming computational language that we used. Next, we talk about the parameter adjustment tests explaining the tested parameters and the chosen values for each version. Finally, the results are compared to an optimal solution computed via complete enumeration and then compared to the best heuristic found in previous studies, namely (Costa 2015).

Finally in Chapter V we present the main conclusions. First we do a recapitulation of what we did during this work, then we show the principal results and at the end we give some ideas/suggestions for further investigations.

Chapter II- Literature Review

As far as we know, there aren't many studies about this topic but, on the other hand, there are several works that study the behavior of metaheuristics and heuristics procedures to conclude how they act and how close they get to the optimal solution.

Some of the works which deal with squared tardiness scheduling with only one machine are (Valente, Schaller and Gonçalves 2013), (Valente and Schaller 2012 [3]) and (Valente and Schaller 2012 [4]). In (Valente, Schaller and Gonçalves 2013) the authors used three metaheuristics, namely iterated local search, variable greedy and steady-state genetic algorithm procedures. With their results they concluded that the studied metaheuristics could achieve optimal solutions for small size problems. Metaheuristics can be very useful as long as the problem is not too large. Variable greedy is often the metaheuristic that comes up with the best solution. Through computational results the authors could conclude that the heuristics used are very efficient in medium size problems.

In (Valente and Schaller 2012 [3]) the authors use a branch-and-bound algorithm to optimally solve small size problems. So they could reduce the size of the problem, the authors used dominance conditions. Dominance rules identify a subset of solutions that contain at least one optimal solution. The results in this work evidence that dominance conditions improve significantly the efficiency of branch-and-bound algorithm procedure.

Another work that deals with scheduling problem for a single machine is (Valente and Schaller 2012 [4]). In this work, authors used dispatching rules to minimize tardiness. This works uses not only heuristics adapted to quadratic objective function as also existing rules for linear problems. With the obtained results it was possible to conclude that heuristics that take into account the quadratic objective show better results that their linear counterparts.

In what regards previous studies that considered earliness and tardiness costs we have (Valente and Schaller 2013) and (Hallah 2014). In (Valente and Schaller 2013) was considered a problem with a set of n jobs and M machines in a permutation flowshop environment as in our work. The authors imposed as main goal to minimize tardiness

and earliness costs using six heuristics. They used simulated annealing (in two strands), neighborhood search, genetic algorithm, variable greedy algorithm and fast ant colony algorithm. Through the results it was possible to conclude that the genetic algorithm achieved a lower total of tardiness and earliness. Thus, the authors concluded that, for this type of problem, the genetic algorithm performed better than the other procedures.

As in (Valente and Schaller 2013), in (Hallah 2014) was considered a permutation flowshop problem where the main goal was also minimize tardiness and earliness costs. In this study, the author opted to use variable neighborhood search combining iterated local search with variable neighborhood descent. "ILS ensures the diversification while VND the intensification." (Hallah 2014). Through the results the author could see that the combination of these two heuristics, iterated local search and variable neighborhood descent, is very efficient not only because of the reduced computational time but also because it is possible to accomplish solutions with great quality.

(Costa 2015) considered the exact same problem as ours: permutation flowshop with a weighted squared tardiness objective. In this work, several dispatching rules were proposed, as well as a three-phase improvement procedure. Through the computational results, the authors concluded that dispatching rules are very efficient and in some cases are the only viable approach for larger instances. In this work it was also possible to verify that specific quadratic heuristic obtain better results than linear procedures.

In (Costa 2015) after getting a set of randomly generated problems, the author applied some heuristics and compared them. After that comparison, improvement methods were applied to the two of best performing rules. To understand if the quadratic heuristic was better than its linear version, the author compared the performance between the two versions using a calculation of the mean relative improvement versus the worst result. The author could conclude that the heuristic adapted to the quadratic objective achieve better results. The better heuristic found in this work with already all the improvement methods applied was QATC_M_NEH_LS which will be the term of comparison to this investigation.

Another approach to this type of problem is explained in (Cheng, Yin, Wen, Lin and Liu 2015), a very recent study. In this article, the authors studied a scheduling flowshop problem but with a little variant, they considered precedence constrains because they believe that this variant happens often in real life situations. However, they only

consider two machines. The goal of this work still was to minimize the total tardiness criterion. First, they tried to find an optimal solution using a branch-and-bound algorithm as it was used in (Valente and Schaller 2012 [3]), which is used in optimization problems that are not very large. Since in this article they only considered two machines, this could be defined as small optimization problem, provided the number of jobs was not large. After they used the branch-and bound algorithm, they looked for a near-optimal solution using a genetic and a larger-order-value method. They concluded that for small optimization problems, the methods used reached a near-optimal solution with good quality and with a short computational time.

(Fernandes and Franinam 2015) is another recent article which studied, once again, a permutation flowshop scheduling problem. This work tries to minimize the makespan subject to a maximum tardiness. A makespan is the total duration of the schedule, that is, the time that all jobs need to finish processing. The authors used a constructive heuristic and a non-population based algorithm. They compared the two methods referenced before to FL (constructive heuristic by Framinan and Leisten) and GA algorithms. They came to conclusion that the constructive algorithm had an excellent performance, and it was tested under certain measures such as the quality of the solutions, the number of feasible solutions, average relative percentage deviation and finally the number of instances with the best solution found.

After analyzing the literature, it is possible to conclude that the better heuristic or metaheuristic depends on the problem and the goal of the study.

Chapter III- Metaheuristics

3.1- Heuristics

The heuristic technique is an optimization procedure that aims to find a solution good enough for a certain problem. There is no guarantee that the solution found is the best one or the optimal solution; however, the solution found will be adequate in most cases. Heuristics can reach a good solution in reasonable time when the perfect solution cannot be found at all or found in satisfactory time. The same heuristic when applied to two different problems can achieve completely different results. The efficiency of one heuristic will always depend on the problem used. There are heuristics that are fast but not so effective and heuristics that are more effective, but with a must higher computational time. Before choosing the heuristic to use, first it is necessary to understand how large our optimization problem is and how close we want to be to the best solution.

3.2- Metaheuristics

On the other hand, a metaheuristic is a heuristic that uses a general strategy to search the solution space, this is, the set of solutions. The general strategy allows applying metaheuristics to any problem since a generic set of steps is followed.

Different metaheuristics use different strategies to search the solution space. Though, all the metaheuristic methods try to overcome the main weakness of local search, namely becoming trapped in a local optimum. A local optimum as the name says is an optimal solution on a specific area and not necessarily the best solution of the problem.

In this work two different metaheuristics will be used, namely iterated local search and steady-state genetic algorithm. These metaheuristics will be explained next.

As it was mentioned before, the metaheuristics also present different results when applied to different problems. There are more effective metaheuristics but with a higher computational time, and metaheuristics with lower solution quality but with a much faster computational time. Before choosing the metaheuristic to be used, it is necessary to combine the processing time (efficiency) and the solution quality (effectiveness) to determine the best one to use on an optimization problem.

3.3- Versions

Before explaining each metaheuristic, some components that are used in the local search and/or generation of an initial solution are described.

Even with different procedures and versions, both metaheuristics used will have the same common stop criterion. Since the main goal of this problem is to minimize the sum of the weighted squared tardiness values, the optimal solution will be definitely a cost equal to zero. So, we can use a maximum runtime as a stop criterion knowing if a solution with a total cost equal to zero is found, the procedure is immediately stopped since there is no better solution than that.

The maximum runtime will increase in both n (number of jobs) and m (number of machines).

3.3.1- API

This version is known as Adjacent Pairwise Interchanges (API). This algorithm works simultaneously with a local search procedure and an adjacent interchanges (with a first-improve strategy) until is not possible to improve.

3.3.2- I+I

The name I+I comes for Interchanges + Insertions. As API does, I+I algorithm works also with a local search procedure but combines it with interchanges and insertion neighborhoods. In similarity to what happens in API, this version applies a first improve strategy to the interchanges and insertion neighborhoods. Until no improvement is found, interchanges are first performed. After that, insertions are used again until no improvement is found. This process of applying interchanges followed by insertions is repeated until no further improvement is possible.

3.3.3- NEH

Finally, the last version is NEH for Nawaz, Enscore and Ham. This procedure was presented by Nawaz M, Enscore Jr EE, Ham I (2008). This procedure takes an initial order of the jobs, and inserts each job in each possible position, choosing the best insertion. In this work, this procedure is slightly modified: we only keep the modified sequence if it is not worse than the initial one. Indeed, and though not common, it is possible for NEH to return a sequence that is worse than the original one. When that happens, we retain the (better) original sequence. Later, this algorithm will be combined with the last two versions: API and II (NEH_API and NEH_II).

3.4- Iterated local search

The metaheuristic presented is called Iterated Local Search (ILS). As indicated on its name, this method applies at each iteration local search to the solution generated.

The pseudo-code for the proposed ILS implementation is given below. And then, all steps will be described and explained.

Procedure: Iterated Local Search

1. Set $(S_{best}, ofv_{best}) = (\emptyset, \infty)$.
2. (S, ofv_S) = Generate_Initial_Solution().
3. If Do_Local_Search(S) == TRUE, set (S, ofv_S) = Perform_Local_Search(S).
4. If $ofv_S < ofv_{best}$, set $(S_{best}, ofv_{best}) = (S, ofv_S)$.
5. While stop criterion is not met:
   5.1. (S_k, ofv_k) = Perform_Kick(S).
   5.2. If Do_Local_Search(S_k) == TRUE, set (S_k, ofv_k) = Perform_Local_Search(S_k).
   5.3. If $ofv_k < ofv_{best}$, set $(S_{best}, ofv_{best}) = (S_k, ofv_k)$.
   5.4. If Perform_Backtrack() == TRUE, set $(S, ofv_S) = (S_{best}, ofv_{best})$.
   5.5. Else, set $(S, ofv_S) = (S_k, ofv_k)$.

In the pseudo-code, S_{best} is the best solution found so far and ofv_{best} is its objective function value. Then we have S and ofv_S which are the current solution and its objective function value, respectively. The same happens in (S_k, ofv_k) but with a twist, the *k.* The letter k is for "kick", basically the same information is provided for the current solution and the kicked solution, the solution obtained by performing a kick on the current solution. This procedure will be explained further.

This implementation is similar to the one used in (Valente, Schaller and Gonçalves 2013) with some differences namely, the initial sequence is different and the neighborhoods/procedures used in the local search are different.

In step 1, the algorithm obviously starts by setting the best solution found so far and the respective objective function value to an empty sequence and infinity, respectively. The next step generates the initial solution.

In step 3 there is a condition that says whether a local search is or not applied to a solution and then, when appropriate, the local search is performed. In this implementation, the local search is always applied when a solution is better than the best found so far. On the other hand, if the solution is not better than the best found so far, the local search is still applied, but with a probability equal to a user parameter $0 \leq ls\_prob \leq 1$. If the current solution is the best found so far, step 4 will update the best solution.

Step 5 has five sub steps. This step keeps the algorithm iterating until a stop criterion is met. As it was explained in chapter 3.3, the algorithm stops if a solution with an objective function value of zero is found or if we reach a defined maximum computation time.

In the sub step 5.1 is possible to see that at each iteration a new solution S_k is obtained as consequence of a kicking procedure on the current Solution S. In this implementation, a kick consists in performing α random swaps and α is a user defined parameter.

Sub steps 5.2 and 5.3 are similar to steps 3 and 4, respectively. Sub step 5.2 determines if a local search is applied, and the next sub step updates the best solution found so far, when appropriate.

The last sub steps 5.4 and 5.5 set a new current solution. If a backtrack is performed, the current solution is set equal to the best solution found so far, otherwise, the kicked solution becomes the new current solution. In this implementation, a backtrack is performed when β consecutive iterations have been performed without improving the best solution found so far, where β is a user defined parameter.

As it was said before, four versions will be used with each metaheuristics to test the efficiency and efficacy of each one. The fours versions used will be presented and described next.

- **ILS_API**
    - Local Search: API
    - Initial Solution: QATC + M + NEH
- **ILS_NEH_API**
    - Local Search: NEH + API
    - Initial Solution: QATC + M
- **ILS_I+I**
    - Local Search: I+I
    - Initial Solution: QATC + M + NEH
- **ILS_NEH_I+I**
    - Local Search: NEH + I+I
    - Initial Solution: QATC + M

The first two versions use API and NEH_API, respectively, in their local search. We are considering these two alternatives, that is, using only API or NEH before API based on (Costa 2015), a work that used dispatching rules for an early/tardy objective. In study paper, the author conclude that applying NEH before some other local search makes the overall procedure faster, that means, the time required by NEH was less than the time it was then saved in the following local search procedure. By using these two versions, we can see if we can obtain the same result, that is, make the entire procedure faster.

In the API version the initial solution is given by the multiple sequence version of QATC (QATC + M) followed by NEH (QATC was the best performing dispatching rule for our problem determined in [14]).

In the NEH_API, the initial solution doesn't include NEH because the initial sequence will always go through local search. Given the current best solution is improved, because so far it is empty, and since local search starts by applying NEH, is not necessary to include NEH in the generation of the initial sequence.

The next two versions, I+I and NEH_I+I, are similar to API and NEH_API, respectively but they use instead I+I in the local search. I+I is indeed slower than API but much more intensive. While testing these four versions, we expect to get a higher ls_prob (probability of applying local search) for API and smaller for I+I.

3.5- Steady – state genetic algorithm

The second metaheuristic used is a steady-state genetic algorithm (SSGA). This metaheuristic is similar is to the generic genetic algorithm , but with a twist, there are no major replacements between generations, as the name "steady-state" implies. At each iteration a new solution is generated by reproduction or mutation but that new solution can be accepted or not. When it is accepted, this new solution will replace the worst solution in the population.

The pseudo-code that we will be using for the proposed SSGA implementation is given below and once again all its steps will be presented and described.

Procedure: Steady-state genetic algorithm

1. Set $(S_{best}, ofv_{best}) = (\emptyset, \infty)$, $(S_{worst}, ofv_{worst}) = (\emptyset, -\infty)$ and $pop = \emptyset$.
2. pop = Generate_Initial_Population().
3. While stop criterion is not met:
   3.1. If Do_Crossover() == TRUE:
      3.1.1. S_1 = Select_Parent(pop).
      3.1.2. S_2 = Select_Parent($pop \setminus S_1$).
      3.1.3. S_{cm} = Perform_Crossover(S_1, S_2).
   3.2. Else:
      3.2.1. S_1 = Select_Chrom_Mutation(pop).
      3.2.2. S_{cm} = Perform_Mutation(S_1).
   3.3. If Do_Local_Search(S_{cm}) == TRUE, set (S_{cm}, ofv_{cm}) = Perform_Local_Search(S_{cm}).
   3.4. If $ofv_{cm} < ofv_{best}$, set $(S_{best}, ofv_{best}) = (S_{cm}, ofv_{cm})$.
   3.5. If $(S_{cm} \notin pop)$ AND $(ofv_{cm} < ofv_{worst})$:
      3.5.1. Replace S_{worst} with S_{cm}.
      3.5.2. Update (S_{worst}, ofv_{worst})

In this pseudo-code, S_{worst} is the worst solution in the current population and ofv_{worst} is its corresponding objective function value.

Similarly, ($S_{1,}$ ofv_1) and ($S_{2,}$ ofv_2) are solutions used in the crossover and/or mutation operations and their respective objective functions values. Also, ($S_{cm,}$ ofv_{cm}) provide the same information for the offspring solution generated via crossover or mutation.

Finally, pop is the set of solutions / chromosomes in the current population.

The algorithm starts by generating an initial population setting the best and worst solutions found so far and creating a so far empty population.

In step 3 and it sub steps the algorithm iterates until the stop criterion is met. At each iteration, a single chromosome is generated using crossover (step 3.1 and its sub steps) or mutation (step 3.2 and its sub steps). The crossover procedure is chosen with a probability equal to a user defined parameter $0 \leq$ cross_prob ≤ 1.

When the new solution is obtained through the crossover procedure, we select two different parents from the population and then we generate the new solution performing the crossover procedure on the chosen parents. Else, a single solution is selected from the population and the new solution is obtained via a mutation procedure. Whether or not a local search is applied to a solution is determined in the same way as in ILS.

Each parent is selected using a probabilistic binary tournament (Goldberg 1990). To decide which pair of individuals we are going do chose, we select two different candidates C_1 and C_2 at random and then, one of these is chosen probabilistically. In this implementation, we used the probability of selecting solution C_1 equal to ofv_{C2} / (ofv_{C1} + ofv_{C2}) where ofv_{C1} and ofv_{C2} are the objective function values of C_1 and C_2, respectively.

A unique offspring solution in the obtained via a uniform order based (UOB) crossover (David 1991). Each position in the sequence is sequentially considered by the UOB crossover, and the corresponding job in the first parent is copied to the offspring with a probability p1_copy_probb. The empty positions are then filled with the missing jobs, in the order in which they appear in the second parent. In this implementation, p1_copy_prob is equal to ofv_2 / (ofv_1 + ofv_2). This means that the probability is proportional to solution quality and the offspring will tend to have more positions copied from the best parent (Beasley 1996).

Finally, with the mutation procedure, we first choose at random a solution and then, a mutated chromosome is then generated from this solution via a gene mutation procedure. In gene by gene mutation, each position in the chromosome is involved in a random move with a given probability. The pseudo-code for the mutation procedure will be presented and described forward.

In the last step of the implementation (3.5), the new solution replaces the current worst member of the population if it is unique (that is, no identical solution is present in the current population) and better than the current worst solution.

The pseudo-code for the mutation procedure is explained next:

Procedure: Fuction Perform_Mutation:

1. Set $i = 1$.
2. While $i < n$:
   2.1. If rand_gen01() < gene_mut_prob:
      2.1.1. Randomly select a position $p \neq i$.
      2.1.2. If (mut_type == INT), swap jobs $[i]$ and $[p]$.
      2.1.3. Otherwise, remove job $[i]$ from its current position and insert it at position p.
   2.2. Set $i = i + 1$.

The user defined parameter $0 \leq$ gene_mut_prob ≤ 1 is the probability of each position being affected by a random move. Also, mut_type $\in$ {INT, INS} is a user defined parameter that indicates whether the random move consists of an interchange (INT) or an insertion (INS) operation.

In step 2, each position in the sequence is considered sequentially. For each position i, the job in the position will be involved in a random move with a gene_mut_prob probability. Step 2.1 and its sub steps perform a random move and when this happens, a different position p is generated at random. If the random move is of the interchange type (INT), the jobs in the two positions are swapped. Else, the job in position i is removed from its current position and reinserted at position p.

Next, we will present the pseudo-code for the generation of the initial population. In the following, (S_{ip}, ofv_{ip}) is a solution created during the generation of the first population and its corresponding objective function value, respectively.

Procedure: Function Generate_Initial_Population()

1. S_{ip} = Generate_Seed_Solution().
2. If Do_Local_Search(S_{ip}) == TRUE, set (S_{ip}, ofv_{ip}) = Perform_Local_Search(S_{ip}).
3. Set $pop = pop \cup S_{ip}$ and update (S_{best}, ofv_{best}) and (S_{worst}, ofv_{worst}).
4. If ofv_{best} = 0, RETURN.
5. Set i_count = 0.
6. While ($\#pop <$ pop_size) OR (i_count < 3 × pop_size):
   6.1. S_{ip} = Generate_Random_Solution().
   6.2. S_{ip} = NEH(S_{ip}).
   6.3. If Do_Local_Search(S_{ip}) == TRUE, set (S_{ip}, ofv_{ip}) = Perform_Local_Search(S_{ip}).
   6.4. If $S_{ip} \notin pop$, set $pop = pop \cup S_{ip}$ and update (S_{best}, ofv_{best}) or (S_{worst}, ofv_{worst}) when appropriate.
   6.5. Set i_count = i_count + 1.
   6.6. If ofv_{best} = 0, BREAK.
7. If ofv_{best} = 0, RETURN.
8. While ($\#pop <$ pop_size):
   8.1. S_{ip} = Generate_Random_Solution().
   8.2. If $S_{ip} \notin pop$, set $pop = pop \cup S_{ip}$ and update (S_{best}, ofv_{best}) or (S_{worst}, ofv_{worst}) as appropriate.
   8.3. If ofv_{best} = 0, BREAK.

In this version, steps 1 to 3 generate a seed solution and then improve it via local search.

If we have an optimal solution, that is, a solution with an objective function equal to zero, the generation of the initial population is stopped in step 4. If not, step 6 (and its sub steps) is in charge of iterating until the initial population has been fully generated, or a maximum of 3 x pop_size iterations have been performed.

In step 6.1 a random solution is first generated and next step 6.2 uses NEH to try to improve that solution. The next sub step 6.3 can apply local search to the same solution. The solution is added to the population if it is unique, and the best or worst variables are updated when appropriate. If an optimal solution with an objective function value of 0 is generated, step 6.6 terminates the procedure immediately. When the local search includes NEH, that NEH step will be skipped inside step 6. Indeed, the solutions generated inside step 6 have already been improved by NEH, so there is no need for the improvement procedure to apply NEH first to these solutions.

To try to prevent a poor initial solution, we introduced the NEH procedure. Indeed, if this procedure was not used, and in the presence of a low user defined ls_prob local search probability, the initial population would consist mostly of randomly generated solutions, which can be of very low quality. If the initial population is quite poor, it can have a negative effect on the performance of a genetic algorithm, which can then require a large number of iterations in order to generate good solutions.

As it was referred before, step 6 finishes if a maximum of 3 x pop_size iterations have been performed without successfully filling the initial solution. This limit is required in order to avoid, in some instances, a quite large number of iterations, or even an endless loop. In fact, in some instances with loose due dates, the application of the NEH procedure may lead systematically to a single or a few solutions in which only a small number of jobs is tardy. Since the objective function value of these solutions is positive, it is not possible to guarantee that the solution found is the optimal although it may be.

In these cases it might take an extremely large number of iterations, or actually be impossible to fill the initial population in step 6 and its sub steps and so, after some experimental tests, the limit of 3 x pop_size iterations was imposed. When step 6 is not succeessfull at filling the initial population, the remaining solutions are generated at random in step 8 and its sub steps. A solution only can be added to the population if it is unique.

As it happened in ILS, here we will be using four versions for the SSGA metaheuristic. All four versions will be presented and described next:

- **SSGA_API**
  - Initial Population:
    - seed solution (only one): QATC + M + NEH
    - remaining (unless limit of 3 * pop_size reached): random + NEH
    - if needed: random
  - Local Search
    - API
- **SSGA_NEH_API**
  - Initial Population:
    - seed solution (only one): QATC + M
    - remaining (unless limit of 3 * pop_size reached): random + NEH
    - if needed: random
  - Local Search:
    - seed solution: NEH + API
    - remaining initial population solutions: API (NEH is applied before)
    - otherwise: NEH + API
- **SSGA_I+I**
  - Initial Population:
    - seed solution (only one): QATC + M + NEH
    - remaining (unless limit of 3 * pop_size reached): random + NEH
    - if needed: random
  - Local Search
    - I+I
- **SSGA_NEH_I+I**
  - Initial Population:
    - seed solution (only one): QATC + M
    - remaining (unless limit of 3 * pop_size reached): random + NEH
    - if needed: random
  - Local Search:

- seed solution: NEH + I+I
- remaining initial population solutions: I+I (NEH is applied before)
- otherwise: NEH + I+I

The main differences between the four versions are the initial solution, the type of local search, and whether or not NEH is included in the local search. If NEH is not included in the local search, the procedure will be applied to the seed solution. But on the other hand, if we include NEH in the local search, we will have the following. First, the seed solution is only QATC + M (since this is the first solution, it will go through local search which includes NEH), and second, NEH is removed from the local search when it is applied to the initial population solutions that already employ NEH, to avoid duplicating NEH.

Chapter IV- Computational results

In this chapter we will first describe all the instances used, as well as the programing language used on the coding process. Then, all the parameters both tested and chosen will be presented. Finally we will see which version was chosen for each metaheuristic and then a comparison of results will be shown.

4.1- Experimental Design

This work uses randomly generated problems. The set of randomly generated problems includes various numbers of of jobs, machines and combinations of due date tightness and range.

The number of jobs has the following sizes: 8, 10, 12, 15, 20, 25, 30, 40, 50 and 75. Regarding the number of machines, we considered 5, 10 and 20 machines. For the processing time p_{ij} of one job in one machine, the number is generated from a uniform distribution over the integers 1 to 100. Finally, the weight w_j is obtained from a uniform distribution [1, 10].

On the other hand, for each job j, an integer due date d_j is generated from the uniform distribution [MS (1- T – R/2), MS (1 – T + R/2)], where MS is an estimate of the makespan calculated using the lower bound proposed in (Taillard1993), T is the tardiness and R is the range of due dates. Both tardiness and range of due dates parameters are set at 0.2, 0.4, 0.6, 0.8 and 1.0.

For each combination of n, M, T and R we created 10 instances at random which totalizes 250 instances for each problem size, where the size is given by both the number of jobs and machines.

Each metaheuristic was ran with 10 seeds which means that we will get, for each metaheuristic, 10 results.

After several experimental tests, we decided to use the following expression for the calculating of the run time stop criterion: $0.2 + 0.0015n^2M$.

For all the computational tests we used a personal computer with a Windows 10 64-bit operating system, an Intel Core i7 4770 3.4G processor and 16GB RAM. The procedures were coded in C++ compiled for 64-bit Windows.

4.2- Parameter adjustment tests

In this chapter we will presented the parameter adjustment tests. First all the tested parameters will be presented and described and then, all the chosen parameters for each version will be also shown.

4.2.1- Tested parameters

ILS

In the ILS metaheuristics, all four versions previously presented require a value for three parameters: α, β and ls_prob. All the corresponding values used for each parameter are in the table below.

Parameter	Values
α	5, 6, 7
β	5, 10, 20, 25
ls_prob	0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

The values considered for α and β are the same used in some previous ILS applications. Since ls_prob values are between 0 and 1, we are considering the entire range, with 0.1 increments.

SSGA

The used parameters for SSGA are different. We used five parameters in this metaheuristic: pop_size, cross_prob, mut_type, and again ls_prob. For each parameter there are a set of values that will be presented in the table below.

Parameter	Values
pop_size	40, 50, 60
cross_prob	0.85, 0.90, 0.95, 0.975
mut_type	INT, INS
gene_mut_prob	0.03, 0.05, 0.07
ls_prob	0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

The parameter adjusment tests were performed on a separate, and smaller, problem set. This set included randomly generated instances with only 10, 20, 30 and 40 jobs. Also, only 5 instances were used for each combination of n, M, T and R. Also, a different maximum runtime was used in the stopping criterion, namely: $0.1 + 0.00075n^2M$.

4.2.2- Chosen Parameters

In this chapter, we will present all the chosen parameters for each metaheuristic. For these tests we only used one seed.

The chosen criterion was based on the speed of the computational time and the quality of the solution. For each heuristic we analyzed all the four versions separately and then, the best combination of parameters of each version were compared to each other so we could reach the best version for that heuristic.

The values chosen for each parameter and each version will be shown next.

ILS

Table 1 - Chosen Parameters for ILS

Version	α	β	ls_prob	stg
API	5	5	1	11
II	5	5	1	11
NEH_API	5	5	0,8	9
NEH_II	6	5	0,6	51

After comparing all results for each version, the values above for all four parameters were the ones that achieved better results taking into account the computational time and the quality of the results. Finally, we compared all four versions and the overall best performed was NEH_II.

SSGA

Table 2 - Chosen Parameters for SSGA

Version	pop_size	cross_prob	mut_type	gene_mut_prob	ls_prob
API	60	0,85	INS	0,05	0,9
II	50	0,975	INT	0,03	0,5
NEH_API	60	0,85	INS	0,07	0,8
NEH_II	50	0,95	INT	0,05	0,3

Once again, each version was analyzed separately and the values for each parameter that achieved the best combination between runtime and quality of solution are presented in the table above. After getting the results for the best values for each version, we concluded that the version I+I was the one that presented the best overall performance.

4.3- Comparison with optimal results

This chapter will present the comparison between the used metaheuristic and the optimal results. As previously remarked, the optimum results were obtained via complete enumeration.

Table 3- Comparison with optimal results

		ivh				n_opt		
m	n	QATC_M_NEH_II	ILS	SSGA		QATC_M_NEH_II	ILS	SSGA
5	8	1,090652153	0	0		201	250	250
	10	1,772517235	0	0		158	250	250
	12	2,285029877	0	0,004679739		135	250	249,5
10	8	0,518990141	0	0		210	250	250
	10	1,719099062	0	0		156	250	250
	12	1,742944584	0	0		124	250	250
20	8	0,430001692	0	0		203	250	250
	10	0,461519121	0	0		172	250	250
	12	1,219551292	0	0		116	250	250

In the table above, ivh represents the improvement the optimum solution provides over each heuristic. It is represented in percentage and it is the average of that improvement for all instances. Also, we present the number of times, for 250 possible, that a (meta)heuristic reached the optimal (n_opt).

Taking the results into account, we can conclude that ILS always achieves the optimal results since the optimum provides no improvement and it obtains the optimal solution 250 times. SSGA does not always achieve the optimal solution but it is optimal for instances with a higher number of machines.

The heuristic used in (Costa 2015), QATC_M_NEH_II, presents worst results than ILS and SSGA.

Table 4- Comparison with optimal results (T and R) – for n=10 and m=10

		ivh			n_opt		
T	R	QATC_M_NEH_II	ILS	SSGA	QATC_M_NEH_II	ILS	SSGA
0,2	0,2	6,725521273	0	0	5	10	10
	0,4	5,4851423	0	0	4	10	10
	0,6	9,917778657	0	0	5	10	10
	0,8	6,172391643	0	0	5	10	10
	1,0	0,682092548	0	0	8	10	10
0,4	0,2	0,822508251	0	0	7	10	10
	0,4	0,674762505	0	0	8	10	10
	0,6	1,867537774	0	0	6	10	10
	0,8	1,618996959	0	0	6	10	10
	1,0	0,1	0	0	8	10	10
0,6	0,2	1,053662799	0	0	7	10	10
	0,4	0,233309714	0	0	6	10	10
	0,6	1,330404155	0	0	7	10	10
	0,8	0,820883879	0	0	4	10	10
	1,0	0,9	0	0	4	10	10
0,8	0,2	0,14570027	0	0	8	10	10
	0,4	0,079655539	0	0	9	10	10
	0,6	0,34978138	0	0	7	10	10
	0,8	0,598363454	0	0	4	10	10
	1,0	0,188192378	0	0	8	10	10
1,0	0,2	1,29122561	0	0	6	10	10
	0,4	0,039570315	0	0	8	10	10
	0,6	0,988010708	0	0	4	10	10
	0,8	0,609435572	0	0	7	10	10
	1,0	0,3	0	0	5	10	10

This table also presents the comparison between the metaheuristics and the optimal results but includes the effect of the T and R parameters. Once again, QATC_M_NEH_II has a worse performance than ILS and SSGA. As we can see, QATC_M_NEH_II is on average 1,72% worse than the optimal solution and it was only 156 times equal to the optimal in 250 possible. QATC_M_NEH_II was mostly better for a lower value of R. QATC_M_NEH_II is better for T equal to 0,8 and worse for T equal to 0,2.

4.4- Comparison of the metaheuristics results

Here we will present the comparison between our metaheuristics and the best heuristic in (Costa 2015), QATC_M_NEH_II (DR). And finally, compare our two metaheuristics with the worst result among themselves.

4.4.1- Comparison with DR (QATC_M_NEH_II)

QATC_M_NEH_II was the best heuristic found in (Costa, 2015) and the comparison between that heuristic and our two is presented in the following table:

Table 5- Comparison with DR

		ivdr		better		equal	
m	n	ILS	SSGA	ILS	SSGA	ILS	SSGA
5	10	1,772517	1,772517	92	92	158	158
	25	7,534408	7,526278	212	212,1	38	37,9
	50	5,931656	6,303622	217	217	33	33
	75	5,463274	5,789758	209,8	210	40,2	40
10	10	1,719099	1,719099	94	94	156	156
	25	9,059383	9,051777	242	242	8	8
	50	10,27416	11,40962	231	231	19	19
	75	10,19207	11,17468	223,7	224	26,3	26
20	10	0,461519	0,461519	78	78	172	172
	25	4,672094	4,67763	244	244	6	6
	50	11,11932	12,88174	250	250	0	0
	75	13,213	14,5856	241,8	242	8,2	8

We only present part of the results, since otherwise the size of the tables would be too large. The first two columns present the average of the improvement between each of the metaheuristic studied and the DR. The results are presented in percentage. For higher number of jobs, the improvement of ILS face DR increases. For 20 machines and

75 jobs, there is a 13,21% of improvement. The same happens for SSGA but with not so high improvements.

Then we have for the third and fourth column (better) the number of times that ILS and SSGA achieve better results than DR. The last two columns show the same logic but for equal results. Both ILS and SSGA are more times better than DR than equal to it.

Table 6- Comparison with DR (T and R) – for n=50 and m=10

		ivdr		better		equal	
T	R	ILS	SSGA	ILS	SSGA	ILS	SSGA
0,2	0,2	17,68051	18,63815	10	10	0	0
	0,4	31,97789	32,52271	10	10	0	0
	0,6	46,09832	48,09381	7	7	3	3
	0,8	12,15574	13,185	2	2	8	8
	1,0	12,96209	12,36277	2	2	8	8
0,4	0,2	12,07914	13,93872	10	10	0	0
	0,4	11,69392	13,59373	10	10	0	0
	0,6	15,70591	18,14881	10	10	0	0
	0,8	13,449	15,64512	10	10	0	0
	1,0	10,69335	11,37388	10	10	0	0
0,6	0,2	7,27057	8,645644	10	10	0	0
	0,4	7,249775	8,967522	10	10	0	0
	0,6	6,572552	8,221712	10	10	0	0
	0,8	7,517961	8,589043	10	10	0	0
	1,0	5,305724	6,351771	10	10	0	0
0,8	0,2	4,734229	5,941428	10	10	0	0
	0,4	4,709278	5,99353	10	10	0	0
	0,6	5,81568	6,82708	10	10	0	0
	0,8	3,503451	4,437535	10	10	0	0
	1,0	3,832228	4,715145	10	10	0	0
1,0	0,2	2,984831	3,669803	10	10	0	0
	0,4	3,52071	4,336147	10	10	0	0
	0,6	3,046977	3,73384	10	10	0	0
	0,8	3,067709	3,659076	10	10	0	0
	1,0	3,226528	3,648596	10	10	0	0

Both ILS and SSGA are always better than DR except for T=0,2. For higher number of R, the improvement increases most of the time. The improvement is higher for T = 0,2. On the other hand, it is lower for T=1,0.

4.4.2- Comparison with worst metaheuristic result

The final comparison is between metaheuristics, namely a comparison of each metaheuristic with the worst combined result. That is, we compare each of the 10 results provided by each metaheuristic with the worst of the combined 20 results.

Table 7- Comparison with worst metaheuristic

		ivh			n_opt	
m	n	ILS	SSGA		ILS	SSGA
5	10	0	0		0	0
	25	0,429336711	0,348986103		16,4	16,3
	50	0,316132083	0,708384796		172,1	187,5
	75	0,452227892	0,813230556		177,4	195,3
10	10	0	0		0	0
	25	0,13018032	0,112597887		25,3	30,4
	50	1,111724639	2,542430069		202,9	222,6
	75	1,280965738	2,366608837		198,9	218,6
20	10	0	0		0	0
	25	0,024985894	0,03142159		15,7	20
	50	1,690337116	4,205906551		223,3	247,9
	75	2,265211691	4,483993594		213,8	236,5

In this table, ivh shows us the average of the percentage of the improvement of the metaheuristics face to the worst result. For example, for 20 machines and 75 jobs, ILS was 2,265% better than the worst result of all. SSGA shows us a better improvement versus the worst result in all of the combinations between the number of jobs and the number of machines.

The last two columns show us how many times ILS and SSGA were better than the worst Thus, for 20 machines and 75 jobs, SSGA was 236,5 better than the worse in 250 possible and 13,5 equal (250-236,5).

Table 8- Comparison with worst metaheuristic (T and R) – for n=50 and m=10

		ivh		n_opt	
T	R	ILS	SSGA	ILS	SSGA
0,2	0,2	1,651590698	2,903909578	9,2	9,8
	0,4	3,568296441	4,453678302	7,2	6,8
	0,6	4,627557954	9,336501615	4,6	4,9
	0,8	2,438059389	5,526126747	1,8	2
	1,0	1,295616138	0,554381831	0,9	1
0,4	0,2	1,486792395	3,578491586	9	10
	0,4	1,349191654	3,490636109	9	10
	0,6	1,43289374	4,313554038	9	10
	0,8	1,832439748	4,34343737	9	10
	1,0	0,608919186	1,364845746	8	8,3
0,6	0,2	0,851715445	2,328202477	9	10
	0,4	0,65879817	2,510078398	9	10
	0,6	0,71987709	2,47308919	9	10
	0,8	0,590305043	1,749461647	9	10
	1,0	0,578010871	1,68089459	9,1	9,9
0,8	0,2	0,575533463	1,836816957	9	10
	0,4	0,49309619	1,834429501	9	10
	0,6	0,537815776	1,60913855	9	10
	0,8	0,462115187	1,426160883	9	10
	1,0	0,468029768	1,383277712	9	10
1,0	0,2	0,321194725	1,025866397	9	10
	0,4	0,402163367	1,244040187	9	10
	0,6	0,278234708	0,984128871	9	10
	0,8	0,272867253	0,881568551	9	10
	1,0	0,29200158	0,728034905	9,1	9,9

Once again, the complete table has a very large size so we are only showing the results for all values of T and R but only for 50 jobs and 10 machines. The first variable, ivh, is the improvement versus the worst result for each metaheuristic. In average, the improvement face the worst result is higher for SSGA and it was also the metaheuristic that was better than the worst more times (222,6 in 250 possible which means that was equal to the worst 27,4 times). In both metaheuristics, the percentage of improvement decreases with the increase of the T value. The average of the improvement for ILS and

T=0,2 is 2,716% and for T=1,0 is 0,31%. The same happens for SSGA, its percentage for T=0,2 is 4,55% and for T=1,0 is 0,97%.

4.4.3- Runtimes

Finally, we will present the computational run times for each metaheuristic studied.

Table 9- Run Times (in seconds)

m	n	ILS	SSGA	max_rt
5	30	1,0812336	0,5508856	17375
	40	4,2441236	3,3818092	30500
	50	7,2184384	8,768428	47375
	75	17,014092	23,277796	105968,75
10	30	3,344592	1,7790996	34250
	40	11,024512	9,6245588	60500
	50	17,580191	23,756299	94250
	75	37,451772	52,857012	211437,5
20	30	6,1139956	2,7264788	68000
	40	23,425442	17,511186	120500
	50	39,038175	49,326003	188000
	75	84,47573	116,97612	422375

The table above presents the computational run times for ILS and SSGA for all m values but only for 30, 40, 50 and 75 jobs. In this table it is also shown the maximum run time computed using this expression: $0.2 + 0.0015n^2m$. For instances with higher number of machines and jobs the computational time is obviously higher since the computational effort is also higher. For 50 and 75 jobs, SSGA is always slower than ILS. Although it has higher computational run time, SSGA achieves better results than ILS.

Chapter V- Conclusion

This work studied a permutation flowshop problem with weighted squared tardiness in order to help a company to save time and money.

We proposed in this study two metaheuristics, namely Iterated Local Search and Steady-State Genetic Algorithm. For each one of them, considered different types of local search and multiple versions.

We first determined appropriate values for the parameters of each version. Then, we determined the best performing version within each metaheuristic The selection criterion was the best combination between the run time and the quality of the solution.

Our results were compared first with the optimal solution, then with DR (proposed heuristic in (Costa 2015)) and finally with the worst result among the metaheuristic results. Both metaheuristics proved to be very efficient for this kind of problem (small to medium size problem), and quite effective, by achieving often the optimal solution. These meta heuristic accomplished also better results than DR since this heuristic is included in our initial solution. The reason why we included this heuristic in our initial solution was first because a poor initial solutions can have a negative effect on the metaheuristic and second, to try achieve no worse results. The computational results showed that ILS was often faster than SSGA but presents results with lower quality. Both of metaheuristic proved to be very efficient.

For future research, others metaheuristics can be analysed, such as variable gready and iterated gready in order to test their performance since metaheuristics are often very efficient in small and medium size problems. On the other hand, additional problem characteristics could be included, such as release dates and setups.

References

Nawaz M, Enscore jr EE, Ham I. (2008) “A heuristic algorithm for the m.machine, n-job flow-shop sequencing problem”. Omega, pp 91-5.

Valente, Jorge and Schaller, Jeffrey and Gonçalves Tomás (2013) “Metaheuristics for the single machine weighted quadric tardiness scheduling problem”. Computers & Operations research, pp 115-126.

Valente, Jorge and Schaller, Jeffrey (2012) “Minimizing the weighted sum of squared tardiness on a single machine”. Computers & Operations research, pp 919-28.

Valente, Jorge and Schaller, Jeffrey (2012) “Dispatching heuristics for the single machine weighted quadratic tardiness scheduling problem”. Computers & Operations research, pp 2223-31.

Valente, Jorge and Schaller, Jeffrey (2013) “A comparison of metaheuristic procedures to schedule jobs in a permutation flow shop to minimize total earliness and tardiness”. International Journal of Production Research, pp 772-9.

M’Hallah, R (2014) “An iterated local search variable neighborhood descent hybrid heuristic for the total earliness tardiness permutation flow shop”. International Journal of Production Research, pp 3802-19.

Costa, Raquel (2015) “Heuristics for a permutation flowshop scheduling problem with weighted squared tardiness”. Dissertation for Master in Modeling, Data Analysis and Decision Support System, FEP.

Sun, XQ and Noble, JS and Klein CM (1993) “Single machine scheduling with sequence dependent setup to minimize total weighted squared tardiness”. Iie Transactions (Institute of Industrial Engineers), pp113-124.

Hoitomt, DJ and Luth, PB and Max, E and Pattipati, KR (1990) “Scheduling jobs with simple precedence constraints on parallel machines”. IEEE Control System Magazine, pp 34-40.

Thomalla, CS (2001) “Job shop scheduling with alternative process plans”. International Journal of Production Economics, pp 125-134.

Taguchi, G (1986) “Introduction to quality engineering: designing quality into products and processes”. Proceedings of the National Electronics Conference, pp 32-39.

Cheng, Shuenn-Ren and Yin, Yunqiang and Wen, Chih-Hou and Lin, Win-Chin and Wu, Chin-Chia and Liu, Jun (2015) “A two-machine flowshop scheduling problem with precedence constraint on two jobs”. Soft Computing, pp 13.

Fernandes-Viagas, Visctor and M. Framinan, Jose (2015) “Efficient non-population-based algorithms for the permutation flowshop scheduling problem with makespan minimization subject to a maximum tardiness”. Computers & Operations research, pp 86-96.

Goldberg DE, Deb K. (1990) “A comparative analysis of selection schemes used in genetic algorithms”. Foundations of Genetic Algorithms, pp 69-93.

Davis L. (1991) “Handbook of genetic algorithms”. New York : Van Nostrand Reinhold.

Beasley JE, Chu PC (1996) “A genetic algorithm for the set covering problem”. European Journal of Operation Research, pp 392-404.